An Analytical Comparison of Some Rule-Learning Programs

AN ANALYTICAL COMPARISON OF SOME RULE LEARNING PROGRAMS

Alan Bundy
Bernard Silver
Dave Plummer

D.A.I. RESEARCH PAPER NO. 215

Paper submitted to *Artificial Intelligence Journal*, 1984.

An Analytical Comparison of Some Rule Learning Programs

by

Alan Bundy, Bernard Silver and Dave Plummer

Department of Artificial Intelligence
University of Edinburgh

Abstract

To become a mature science, Artificial Intelligence needs more theoretical work. One form this should take is the analytic comparison of existing programs to extract precise techniques from the code, compare similar techniques, expose faults, and extend successful techniques.

In this spirit, we compare the rule learning programs of Brazdil, [2], Langley, [7], Mitchell et al, [14, 15], Shapiro, [18], and Waterman, [21]. Each of these programs has two main parts: a critic for identifying faulty rules and a modifier for correcting them. To aid comparison we describe the techniques of the various authors using a uniform notation. We find several similarities in the techniques used by the various authors and uncover the relations between them.

We compare the rule learning programs with the concept learning programs of Quinlan, [17], and Young et al, [23]. The two types of program have much in common, and many of the rule modifying techniques are subsumed by the techniques of Young et al. Quinlan's program is able to learn disjunctive concepts that are more general than those that can be learned by most of the other programs.

Keywords

Learning, concept learning, rule learning, production systems, PROLOG, Generalization, Discrimination, Version Spaces, Focussing, Classification.

Acknowledgements

We could not have conducted this comparison without some clarificational conversations with the surveyees, namely: Ranan Banerji, Pavel Brazdil, Pat Langley,

Tom Mitchell, Gordon Plotkin, Udi Shapiro and Richard Young. We thank them for their time and patience. We also got valuable feedback from some informal seminars given in the Edinburgh AI department.

This work was supported by SERC grants GR/B/29252 and GR/C/20826, and SERC studentships to Dave Plummer and Bernard Silver.

1. Introduction

Artificial Intelligence is a young science, which is still developing its methodology and its terminology. Currently, there is no universal agreement about many technical terms. Some techniques have several different names or none, and some names refer to several different techniques. Some techniques have not yet been abstracted from the programs in which they were originally developed. The relationship between techniques is not well understood.

To correct this state of affairs, it is necessary to analyse existing AI programs and techniques:

- to abstract techniques from programs by explaining them in a code free manner;
- to identify the range of applicability of these techniques;
- to locate and repair flaws in these techniques;
- to establish the relationship between techniques for the same task; and
- to extend the range of existing techniques.

We call such research **analytic comparison** .

This paper is an analytical comparison of the following work in the area of AI learning programs:

- The ELM program of Brazdil, [1, 2], which transforms a specification into a program in the domains of: simple arithmetic, algebra and letter series

completion.

- The AMBER program of Langley, [7], which acquires the ability to generate simple English utterances.

- The LEX program of Mitchell et al, [14, 15], which acquires heuristics in the domain of symbolic integration.

- The extension by Quinlan, [17], of the concept learning system, CLS, of Hunt et al, [5]. Quinlan's program, ID3, was used by its author to classify certain chess positions as lost or won.

- The Model Inference System of Shapiro, [18], which synthesizes logic programs from examples in the domains of arithmetic, list processing, etc.

- The P program of Waterman, [21], which acquires heuristics for betting in the game of draw poker.

- The extension by Young, Plotkin and Linz, [23], of Winston's concept learning program, [22], which learns the definitions of simple structures, e.g. an arch, from examples and near misses.

It seemed to us that the above listed researchers had provided, sometimes complementary, sometimes alternative, techniques for solving isomorphic problems, but that this was obscured by their use of different formalisms and terminology.

Our comparison is analytical in that in order to clarify the similarities and differences between the techniques we have described them with a uniform formalism and terminology. To keep the comparisons simple we have suppressed some of the details of the techniques, but we hope we have retained their spirit. We have also suppressed all domain specific aspects of the techniques, except for the use of domain specific rules in our worked examples, and even here we have deliberately applied one person's technique to another's rules.

In this comparison, we include only those programs that are of direct relevance to rule learning, and that are representive of key techniques. This differs from the

approach of non-analytical surveys, for example that in the Handbook of Artificial Intelligence, [4], which attempt to cover a very wide range of diverse learning programs, describing each in the terminology of the program's author. Our approach differs from that of Smith et al, [20], as that survey concentrated on the architecture of learning programs, rather than techniques.

Furthermore, we have avoided a behavioural classification of learning, e.g. "rote learning", "learning by being told", "learning from examples", "learning from analogy", such as that found in [4]. We have tried to develop an algorithmic classification. In our experience, programs in two different behavioural class may contain similar algorithms for some subtask, and there are often several different algorithms for each behavioural class. For instance: both "learning by being told" and "learning from examples" programs may use the critic procedures described in section 3; and the two, very different, techniques of Version Spaces , [14], and Precondition Analysis, [19], can be used for "learning from examples". This makes the behavioural classification less than helpful to the AI researcher whose main task is to design algorithms.

2. The Learning Task

The programs surveyed are usually considered to fall into one of two groups. The programs of Quinlan and Young et al are **concept learning** programs, all the others are **rule learning** programs. These two classes are in fact closely related: concept learning is contained within rule learning.

2.1. Concept Learning Programs

A concept learning program has to learn a symbolic description that enables it to determine whether or not an object is an instance of the target concept. Perhaps the most famous example of a concept learning program is that of Winston, [22], that can learn the concept of an arch. The program of Young et al, [23] is an extension of Winston's.

2.2. Rule Learning Programs

The task tackled by the rule learning programs is to modify a set of rules of the form **hypothesis** implies **conclusion** , i.e.

$$H \rightarrow C \qquad (i)$$

These programs often use concept learning techniques to modify the hypotheses of such rules.

The case we will consider first is that the hypothesis H is a conjunction

$$H_1 \ \& \ \ldots \ \& \ H_n \qquad (ii)$$

where each of the H_is is a negated or unnegated atomic formula. We will call such rules **conjunctive rules** and each H_i a **condition** of the hypothesis. In logic, conjunctive rules are called **Horn Clauses** .

Secondly, we will consider a more general class of rules in which the hypothesis, H, may be made from conjunctions *and* disjunctions of negated or unnegated atomic formulae. We will call such rules **disjunctive rules** . Disjunctive hypotheses can always be put in conjunctive normal form, as a single conjunction of disjunctions, or in disjunctive normal form, as a single disjunction of conjunctions.

All the programs we consider are able to learn conjunctive rules, whereas only a few can learn disjunctive rules. We will mainly be concerned with conjunctive rules. Disjunctive rules will be discussed in section 6.

For each set of rules there is some target behaviour, i.e. how it should behave once learning is complete. The rules are modified until this target behaviour has been achieved.

Some example rules are given in figure 2-1. In the figure, and throughout this

paper, we adopt the PROLOG[1] convention that identifiers beginning with a capital letter denote variables, and those beginning with a lower case letter denote constants.

In the case of Brazdil and Shapiro the rules are PROLOG clauses, which are run in backwards chaining mode by the PROLOG interpreter. Consider the first rule in figure 2-1. If the current goal unifies (matches) against

(X5 + X6) + X2 = X3,

the PROLOG interpreter attempts to solve the subgoals

X5 + X6 = X4, and

X4 + X2 = X3.

In the case of Langley, Mitchell et al, and Waterman the rules are production rules, which are run in forwards chaining mode. Consider the second rule in figure 2-1. If the relations

describe(X) & object(X,Y) & ~definite(X) & singular(X)

hold for the current state, the production system asserts

prefix(X,a).

This can be interpreted as AMBER saying "a", before X, where X is the indefinite, singular, object of some event Y.

2.3. Types of Fault

For our purposes it is necessary that the rules have a truth value. It will be convenient to consider the rules as being formulae of Predicate Calculus, with a truth value assigned by a standard model using Tarskian semantics.

---

[1] A description of PROLOG can be found in Clocksin and Mellish, [3].

A rule, subs1, for addition in Peano arithmetic, from Brazdil, [2].

$$X5 + X6 = X4 \;\&\; X4 + X2 = X3 \rightarrow (X5 + X6) + X2 = X3$$

whose procedure interpretation is:

> To add three numbers, add the first two and then add the result to the third.

A rule for language generation, from Langley, [7].

$$describe(X) \;\&\; object(X,Y) \;\&\; \neg definite(X) \;\&\; singular(X) \rightarrow prefix(X,a)$$

which can be paraphrased as:

> If you want to describe X and X is the object of Y and X is not definite and X is singular then prefix X with "a".

Figure 2-1: Some Example Rules

The rules are modified because they contain a fault. These faults can be of two types:

- **Factual faults** : A rule is false, i.e. the rules constitute a program which calculates incorrect answers.
- **Control faults** : The rules are true, but have undesirable control behaviour when run as a program, e.g. they do not terminate.

A rule which is overly general will either be false in the standard model, in which case it has a factual fault, or it will be true, in which case it has a control fault.

We assume throughout that the rules can be modified into a consistent set that successfully accounts for all the data. In particular, we will not discuss how the programs deal with data that is noisy, or inconsistent for some other reason. Some of the authors do discuss this, see Mitchell's [11] for example.

2.3.1. Factual faults

An example of a factual fault is

describe(X) & object(X,Y) & plural(X) -> prefix(X,a).

This rule can be paraphrased as

> If you want to describe X and X is the object of Y and X is plural, then prefix X with "a".

The rule is incorrect, plural objects should not be prefixed with "a", this constitutes a factual fault.

2.3.2. Control faults

As an example of a control fault, consider the following rule, modified from Brazdil.

X1 + X2 = X3 -> X1 + X2 = X3

This is factually correct, but if used in its current form is likely to cause a loop. However, if weakened to the rule:

X5 + X6 = X4 & X4 + X2 = X3 -> (X5 + X6) + X2 = X3

it will not cause a loop, and defines a useful procedure for specifying the order in which addition is to be evaluated. The weakening is effected by instantiating X1 to (X5 + X6) and replacing (X5 + X6) + X2 = X3 by X5 + X6 = X4 & X4 + X2 = X3.

It is a common device in rule based programming to impose control on a program by weakening a rule by instantiation or by giving it a stronger hypothesis. Such weakening can also turn a factual incorrect rule into a correct rule. This is why the same learning techniques are often applicable to both factual and control faults.

2.4. The Main Control Loop

The faults in Langley and Shapiro's rules were factual and those in Mitchell et al's rules were control faults . Waterman and Brazdil considered both types of fault.

The programs listed, except for Quinlan's and Young's, above all used the following main control loop.

Until the rules are satisfactory:

1. **Identify a fault with a rule;**
2. **Modify the rule to remove the fault.**

Note that the modification of the rule should not introduce new faults!

Some of the authors also tackled the problem of creating new rules, as opposed to modifying existing ones. This is discussed in section 5.

Following Smith et al, [20], we will call the subprogram responsible for identifying faults the **critic** . We will call the subprogram responsible for modifying the rules the **modifier** . In the next section we consider the criticism techniques used by each of the above researchers and in the following section we consider the modification techniques.

3. Criticism Techniques for Credit/Blame Assignment

All the programs we are comparing identify faults by running the existing rules on a problem and then analysing the resulting **rule trace** . The analysis must identify where the rules behaved correctly, called **positive training instances** by Mitchell et al, and where they behaved incorrectly, called **negative training instances** . Both sorts of information can be used: the positive instances to generalize the rules and the negative instances to correct them.

Negative instances can be of two types.

- **Errors of commission** : A rule fired incorrectly, because it was insufficiently constrained. This may produce a factual fault, because the rule produces an incorrect result, or it could lead to undesirable behaviour, i.e. a control fault.

- **Errors of omission** : A rule failed to fire, either because it was incorrectly constrained, or the required rule simply does not exist. In the first case the error could be due to either a factual or control fault.

The modifier requires three pieces of information on each instance.

- The type of instance: positive, negative-commission or negative-omission.

- The rule.

- The context, consisting of the variable bindings when the rule was fired.

Following Brazdil, we will adopt the convention that the variable bindings of positive instances are called the **selection context** and the variables bindings of negative instances are called the **rejection context** .

One of the purposes of a critic is to solve the **credit assignment problem** . This term was coined by Minsky, [10], to describe the problem of deciding which rules are responsible for certain aspects of the program's behaviour, desirable and otherwise.

In the following section we describe some criticism techniques for identifying control faults and factual faults .

3.1. Using Ideal Traces

One very common critic technique, in the programs we compared, is the use of an **ideal trace** , i.e. an account of what rules *should* have fired and in what sequence. This technique is appropriate for finding both control and factual faults, and is the only technique used by the above programs to find control faults . Some of the programs take the ideal trace as input, others work it out by analysis

using problem solving and inference techniques. The detailed study of these techniques lies outside the scope of this paper, but we describe Mitchell et al's technique in the next section to illustrate the general idea.

The ideal trace is compared with the actual trace of the rules, the rule trace , to locate the first point at which the traces differ. This enables the faulty rules to be identified.

Running the rules causes a search tree to be grown (see figure 3-1). The rule trace is a path through this tree. If this trace differs from the ideal trace it is because, at some point, the rule that fired in the rule trace, R_r, differs from the rules that fired in the ideal trace, R_i. R_r exhibits an error of commission and R_i exhibits an error of omission.

The question then arises of whether the current set of rules contains R_i. If not, it must be created using the techniques described in section 5. In this section, we deal with the case where R_i is in the current rule set.

Correcting the error of commission will (eventually) correct the error of omission. When the error of commission is corrected, R_r no longer fires. If another rule, R'_r, now causes an error of commission, this will be corrected. Eventually, R_i must be the most preferred rule, correcting the error of omission. We are therefore free to concentrate on errors of commission .

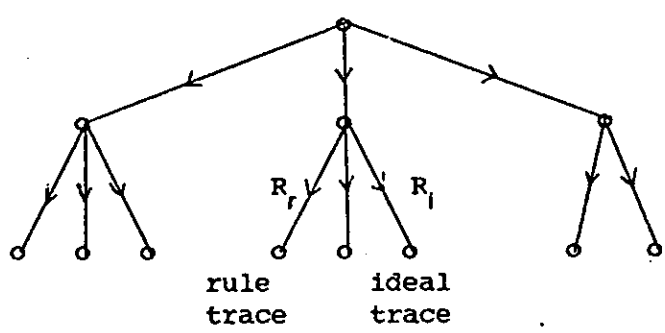


Figure 3-1: Search Tree for Program's Rules

The technique can be summarised as follows:

(a) Grow the rule trace by running the rules on a problem.

(b) Compare with the ideal trace and find the first place at which they differ.

(c) The rules which fired before this point, together with their associated selection contexts, are positive training instances .

(d) The program rule which fired at this point, together with its associated rejection context , is a commission error.

For instance, suppose the rule

subs: X1 = X4 & X4 + X2 = X3 -> X1 + X2 = X3

fires in the context (3/X1, 2/X2, Ans/X3) but that the rule

subz: X2 = X4 & X1 + X4 = X3 -> X1 + X2 = X3

is fired in the ideal trace, then subs in the context (3/X1, 2/X2, Ans/X3), is an error of commission.

3.2. Constructing the Ideal Trace

Both Brazdil and Mitchell et al use their programs to produce the ideal trace , but the approaches differ in an important way.

In Brazdil's program, the ideal trace is constructed by a subprogram that is essentially separate from the rest. It uses only correct rules, and applies them only when appropriate. It exhibits the behaviour that the learning component is aiming towards. The learning component has no access this subprogram. The learning process would be the same if the user supplied the trace. Brazdil's subprogram essentially corresponds to the target rules. The assumption that such a subprogram exists will rarely be appropriate. Brazdil's 'solution' should be considered to be ad hoc scaffolding which enables the rest of the program to run.

Mitchell et al's program constructs its ideal trace by pruning the rule trace. We will call this technique **Solution Extraction** . The basic idea is to find a desirable branch of the program's search tree, and prune away all other branches. In the simplest case the desirable branch will be any branch leading to a solution. Mitchell et al go further and try to find a least cost solution. Solution Extraction does not assume access to the target rules.

The rules used by the program are only partially specified (see section 4.3). A numerical score is assigned to how well they apply in a situation and this score is used as the evaluation function in a heuristic search. A resource limit is given to the problem solver, which puts an upper bound on the amount of c.p.u. time and memory it may use in attempting to solve a problem. These limitations may prevent the program finding the least cost solution and so lead to an erroneous ideal trace . To mitigate this a further expansion is made of negative training instances before they are finally sent to the rule modifier . This still doesn't guarantee that the least cost solution has been found.

3.3. Contradiction Backtracing

In this section we consider Shapiro's technique for locating factual faults . This technique is called **Contradiction Backtracing** . It is not suitable for finding control faults .

Suppose that the current rule set implies P, but that P is known to be false. The falsity of P may be given by the program user or calculated from the standard model. Clearly, at least one of the current rules is factually faulty, but we may not be able to tell which one from the model.[2] If the faulty rule contains free variables and the model has an infinite domain then an infinite series of instances must be considered. Contradiction Backtracing uses the rule trace and the

---

[2] This is another instance of the credit assignment problem .

application of the model to variable free rules, to identify the faulty rule. Since the model is only used to test a finite number of variable free rules, Contradiction Backtracing always terminates.

Firstly ~P is added to the set of rules. As P was implied by the original rule set, the new rule set is inconsistent. The empty clause can therefore be derived by resolution. Once this derivation has been obtained, Contradiction Backtracing uses it in reverse. Starting from the empty clause, the process makes use of the model (see below) to discover which parent of each resolution step is false. Eventually, the false parent must be a rule, and this rule therefore has a factual fault.

The technique can be summarised as follows:

(a) Add ~P as a new rule. (The rules are now inconsistent.)

(b) Derive ->, the empty clause, from the rules by resolution. (The derivation is a rule trace, but unlike previous techniques we will not need an ideal trace.)

(c) Set -> to be the current clause of the derivation and () to be the accumulated substitution.

(d) Until the current clause is a rule, do the following:

  (i) The current clause was derived by resolving clauses, C and D, with unifier Φ. The proposition K, from C, and negated proposition L, from D, were resolved away, where $\sim K\Phi \equiv L\Phi$.[3] Apply the accumulated substitution to $K\Phi$ to form Q.

  (ii) If Q contains any free variables then instantiate it to a variable free proposition, Q', in any way, using the substitution Θ.

  (iii) Form a new accumulated substitution by combining it with Φ and Θ.

---

[3] $\equiv$ means 'is syntactically identical to'.

(iv) If Q' is true then let D be the current clause.

(v) Otherwise Q' is false. Let C be the current clause.

(e) The current clause is a faulty rule, and applying the accumulated substitution to it gives a false instance.

The decision as to whether each Q' is true or false can either be supplied by the program user or calculated from the standard model. Note that the only calls on the model are to decide the truth value of formulae without free variables (or quantifiers). Shapiro uses the term **ground oracle** to describe something (e.g. model or user) that can determine the truth value of such formulae.

Note also that the instantiation of Q to Q' will not be necessary if Q is variable free. Different choices of θ may lead to different faulty rules, and may all be tried.

For instance, suppose the current rule set were:

describe(Y) & object(X,Y) -> describe(X)

describe(X) & object(X,Y) -> prefix(X,a)

-> object(balls,event2)

-> describe(event2)

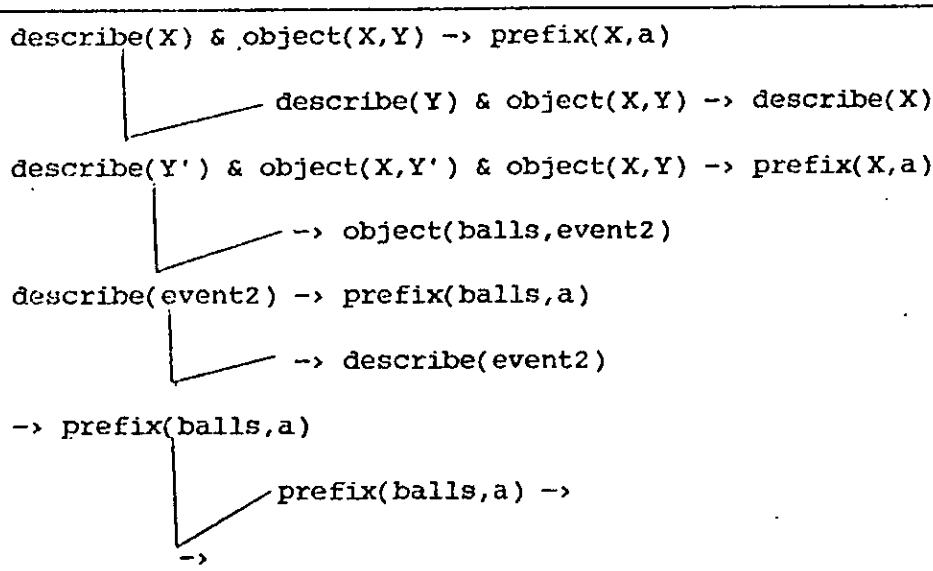
but that prefix(balls,a) were known to be false. This might correspond to the child having made the utterance: "The dog chases a balls". Adding the new rule

prefix(balls,a) ->

we can derive the empty clause with the derivation given in figure 3-2. The Contradiction Backtracing algorithm now goes through the steps tabulated in table 3-1. The rule

describe(X) & object(X,Y) -> prefix(X,a)

has now been identified as faulty, with substitution {balls/X, event2/Y} giving a false instance.



Current Clause	Q'	Truth Value
->.	prefix(balls, a)	false
-> prefix(balls, a)	describe(event2)	true
describe(event2) -> prefix(balls, a)	object(balls, event2)	true
describe(Y') & object(X, Y') & object(X, Y) -> prefix(X, a)	describe(balls)	true
describe(X) & object(X, Y) -> prefix(X, a)		

Table 3-1: Backtracing Through a Contradiction

3.4. Summary

The critic procedures establish the existence of a fault in the rules and locate the faulty rule, the substitution that made it faulty and what type of fault it is. They also identify circumstances in which a rule was correctly fired. They feed to the modifier procedures: the type of each instance, the rule in question, the context in which it fired.

The major critic technique, used by nearly all the programs compared, and applicable to both control and factual faults , is comparison of the actual rule firings

with an ideal trace .

1. Some programs have this ideal trace directly input.

2. some have it provided by a program which corresponds to the target state of the rules being learned, and .

3. some work out the ideal trace by a process of Solution Extraction.

(2) is just a convenient automation of (1), and neither is interesting as a learning technique. We considered only one example of (3), the LEX problem solver, although others may be found in the literature.

4. Modification Techniques for Conjunctive Rules

Once a fault has been located, the faulty rule can be modified. The following modification techniques for conjunctive rules , were used in the programs we compared.

1. Ordering the rules, e.g. specifying that

   H -> C should always be fired in preference to H' -> C'

   This technique is strictly only appropriate for control faults and was so used by Brazdil and Waterman. However, Langley also used it to suppress factual errors.

2. Instantiating a rule, e.g. transforming

   H(X,Y) -> C(X,Y) to H(X,X) -> C(X,X)

   This technique is appropriate for both factual and control faults and was used by Brazdil and Shapiro.

3. Adding extra conditions to a rule's hypothesis, e.g. transforming

   H -> C to H & H' -> C

   This technique is appropriate to both factual and control faults, and was used by Brazdil, Langley, Shapiro and Waterman.

4. Updating a rule's hypothesis to take account of new instances, e.g. transforming

H -> C to H' -> C

where H' is derived from H by concept learning . This technique is appropriate to both factual and control faults, and was used by Waterman and Mitchell et al.

Methods 2 and 3 both make the modified rule more specific. Method 4 narrows the range of uncertainty about the rule.

The techniques described above modify conjunctive rules . The techniques of Mitchell and Langley have a limited capacity to learn disjunctive concepts, which is dependent on a favourable training order. Of the programs compared in this paper, only Quinlan's program learns disjunctive concepts flawlessly. Modification techniques for disjunctive rules are described in section 6.

4.1. Ordering the Rules

All rule based systems need a control strategy to decide conflicts between two or more applicable rules. If a system uses a priority ordering on the rules then control faults can often be corrected by re-ordering the rules. In this section we explain the ordering technique used by Brazdil.

Brazdil's system started with an unordered set of rules, and imposed the partial order required to keep the rule trace in line with the ideal trace . His critic and modifier worked as co-routines, discovering conflicts and resolving them by imposing an order. The technique can be summarised as follows:

(a) Suppose that rules, $P_1, \ldots, P_n$, are applicable and that rule P_i is fired in the ideal trace. In the following $P > Q$ means that the system will fire P before Q.

(b) If $P_j > P_i$ for $i \neq j \in (1, \ldots, n)$ then create a new rule P'_i from P_i, by techniques to be described in subsequent sections, and impose the order $P'_i > P_j$ for all j, such that $i \neq j \in (1, \ldots, n)$.

(c) Otherwise impose the order $P_i > P_j$ for all j, such that $i \neq j \in (1, \ldots, n)$.

For instance, suppose the rules:

```
subs:   X1 = X4 & X4 + X2 = X3  ->  X1 + X2 = X3
subz:   X2 = X4 & X1 + X4 = X3  ->  X1 + X2 = X3
eq:      -> X1 = X1
```

are all applicable, but that the ideal trace records that subz should fire, then the orders

subz > subs and subz > eq

will be imposed.

If at some later stage the same rules are in conflict, but the ideal trace records that subs should fire, then we cannot impose the order subs > subz because this would contradict the existing order subz > subs. In this case a new rule, subs1, is built from subs and the orders

subs1 > subz and subs1 > eq

are imposed. Since > is transitive these new orders also imply that subs1 > subs. The techniques for making subs1 are described in the next two sections.

Langley uses rule re-ordering to deal with factual faults . Rules are ordered by having an associated priority number. Faulty rules have their priority reduced so that they are less likely to fire in future. Consequently, the same fault may be redetected several times before the rule's priority drops so low that it is never selected. In [7], Langley justifies this strange technique with a rather dubious psychological argument. In a personal communication, however, he points out that the technique allows AMBER to learn disjunctive concepts, see section 6.4, and to learn from noisy data.

In certain situations Waterman's program is unable to modify the existing rules successfully. In this case the program obtains the correct rule from the trainer. The program then has to assign a priority to the rule to ensure it fires only when needed. Waterman does not report how his program deals with the problem of

conflicting priority. Perhaps to try to avoid the problem, this technique is only used as a last resort.

Of these rule ordering techniques, Brazdil's seems the neatest. He explicitly records the partial order which is forced by the critic information. No unnecessary orderings are imposed, as with the total orders of Langley and Waterman. Hence, maximum use is made of ordering, and there is no unforced use of alternative modification techniques.

4.2. Adding Extra Conditions to a Rule's Hypothesis

In this section we will consider how a rule can be modified by adding an extra condition to its hypothesis.

Suppose a rule, H -> C, has given a commission error, but that this rule has been applied correctly in the past. The variable bindings of the correct application will give us a selection context and the variable bindings of the incorrect application will give us a rejection context . The idea of this technique is to find some difference between the selection and rejection contexts and use this difference as the new condition. The technique is realised in what, following Langley, we will call **Discrimination** .

(a) Apply the selection and rejection context substitutions to a fixed set of literals,[4] called the **description space** .

(b) Find a literal, H', which is true in the selection context and false in the rejection context. H' is called a **discriminating literal** .

(c) Form the new rule H & H' -> C.

The new rule is only applicable to the selection context .

For instance, suppose the rule

[4] A literal is either a proposition, e.g. P(X), or a negated proposition, e.g. ~P(X).

describe(X) & object(X,Y) -> prefix(X,a)

has been correctly applied to the word 'ball' and incorrectly applied to the word 'balls'. We have:

Selection Context: {ball/X, event1/Y}

Rejection Context: {balls/X, event2/Y}

To find the difference, H', between these contexts we apply them as substitutions to the literals in the description space :

singular(X), ~singular(X), definite(X), ~definite(X)

The only discriminating literal is singular(X). Adding this to the rule as a new condition yields:

describe(X) & object(X,Y) & singular(X) -> prefix(X,a)

In section 4.2.2 we will describe a special case of Discrimination, and in section 4-3 we will generalize Discrimination to a more powerful technique.

4.2.1. Far Misses

In the example above, this particular combination of selection context , rejection context and description space , yields only one discriminating literal. Following Winston we call such a situation a **near miss** . If there is more than one discriminating literal then we will call the situation a **far miss** . A far miss would arise if we added to the description space the literal, past(Y), meaning event Y happened in the past. If past(event1) was true but past(event2) was false then past(Y) would also be a discriminating literal for the above contexts, and there would be a choice of new rules to form. Clearly the description space is of pivotal importance in determining whether a discriminating literal is found and what sort of new rules are formed. In all the programs considered here the description space is user supplied, and it is difficult to see how it could be otherwise.

Waterman deals with far misses by demanding extra information from the user

which will settle the ambiguity (see section 4.3).

Langley deals with far misses by creating a new rule for each discriminating literal, e.g.

```
describe(X) & object(X,Y) & singular(X) -> prefix(X,a)
describe(X) & object(X,Y) & past(Y) -> prefix(X,a)
```

Any useless creations (like the past rule) would eventually be criticised as faulty and fall low in the priority ordering.

Brazdil deals with far misses by including all the discriminating literals in a disjunction, e.g.

```
describe(X) & object(X,Y) & (singular(X) v past(Y))
        -> prefix(X,a)
```

He uses a modified version of Discrimination which tries to prune such disjunctions before adding new conditions. For instance, if the following contexts arise:

Selection Context: {ball/X, event1/Y}

Rejection Context: {balls/X, event3/Y}

where past(event3) is true then Brazdil's Discrimination algorithm drops past(Y) from the disjunction to form the rule:

```
describe(X) & object(X,Y) & singular(X) -> prefix(X,a)
```

In section 6.4 we discuss how the Langley and Brazdil far miss techniques can be used to learn disjunctive concepts.

4.2.2. Instantiating a Rule

An alternative to adding an extra condition to a rule is to instantiate it. This is really a special case of adding an extra condition, but can lead to more efficient rules since the extra condition is handled by the pattern matcher. For instance,

suppose we are modifying the rule:

subs: X1 = X4 & X4 + X2 = X3 -> X1 + X2 = X3

in the contexts

Selection Context: ((3+1)/X1, 1/X2)

Rejection Context: (3/X1, 2/X2)

and the description space contains X1 ≡ X5 + X6. X1 ≡ X5 + X6 is a discriminating literal , so we could add it as an extra condition . Alternatively, we could instantiate the rule with the substitution, ((X5+X6) / X1) to form

subs1: X5 + X6 = X4 & X4 + X2 = X3 -> (X5 + X6) + X2 = X3

Instantiation with the substitution (t/X) is always an alternative when the discriminating literal is X ≡ t, for some variable X and term t.

4.3. Concept Learning of the Rule Hypothesis

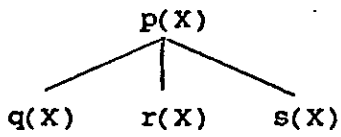
In this section we consider how a rule can be modified by updating its hypothesis using concept learning techniques, like those used by Winston, [22], for learning the concept of an arch from examples and near misses. This technique can be regarded as a natural extension of the one described in the last section. This relationship is most clearly seen by considering the technique of Young et al, because it generalises much of the Winston and Brazdil/Langley/Waterman techniques, and is similar to, but more easily explained than, the Mitchell et al technique.

We therefore adopt the strategy of explaining first the Young et al technique, pointing out the differences from the other techniques as we go. We will be defining an algorithm which we will call Focussing . We describe Quinlan's technique, Classification , for learning disjunctive concepts in sections 6.5 and 6.6, and compare Classification with Focussing in section 6.7.

4.3.1. The Description Space

The description space , in Focussing, tries to capture the notion of a partially specified concept, in which some situtations are know to lie outside the concept, some inside, and some, in a grey area, are yet to be decided. The Focussing process works to reduce this grey area.

The description space consists of a set of **relation trees** (see figure 4-1). Each node of the tree is labelled with a relation; relations in the same tree being applied to the same arguments. The label of the root node is the relation which is always true. The label of a node is logically equivalent to the exclusive disjunction of the labels of its daughters, i.e. the arrangement



implies that

$$p(X) \leftrightarrow (q(X) \dot{\vee} r(X) \dot{\vee} s(X))$$

where $\dot{\vee}$ means exclusive or. Thus, any given instance will cause the relation labelling exactly one tip node to be true. The instance is said to **specify** this relation.

These trees make explicit the relationship between a proposition and its negation by arranging them as the labels on the two daughters of the root node. A tree consisting only of a root node with two daughters will be called a **minimal tree** . The singular/plural and definite/indefinite trees of figure 4-1 are minimal.

This description space allows a *partially specified* rule hypothesis to be represented. During the course of rule learning this partially formed hypothesis is gradually firmed up until it is *completely specified*. The partial representation is achieved by placing two markers in each tree: an **upper mark** and a **lower mark** as in figure 4-1. The partially specified rule is represented by the rule conclusion and

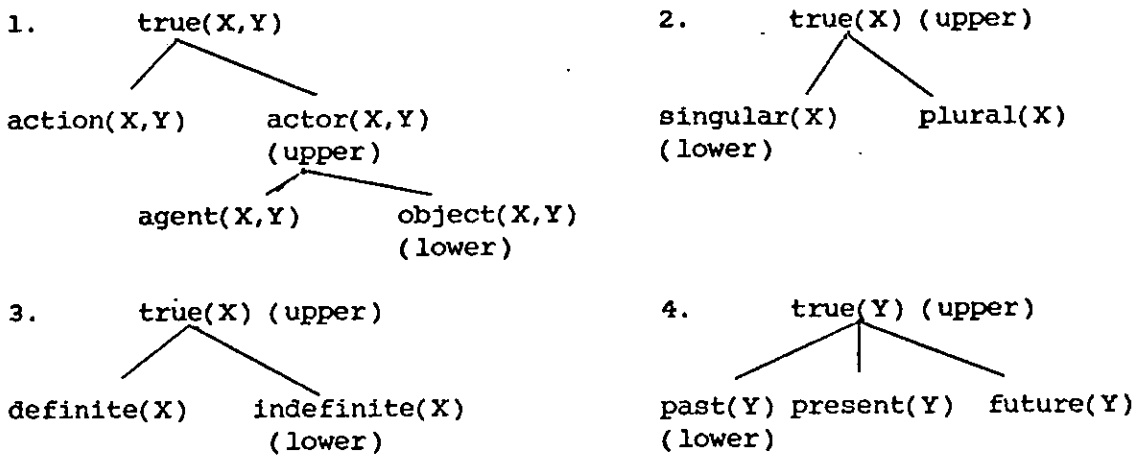


Figure 4-1: Description Space for Language Acquisition

the description space together with its marks. We will call this a **rule shell** .

Any relation above the upper mark[5] is outside the concept, e.g. action(X,Y) and true(X,Y). Any relation in the tree below the lower mark is inside the concept, e.g. object(X,Y). Any relation between the upper and lower marks is in a grey area, about which the program is not sure, e.g. agent(X,Y) and actor(X,Y). The condition is **firmed up** when the upper and lower marks coincide. The rule shell is firmed up into a rule when each of its conditions is firmed up. Focussing works by moving the upper marks down and/or the lower marks up, until they coincide.

Since the rule shell only partially specifys the rule, there is some ambiguity about what rule to use when forming rule traces. In particular, we can take two extreme views:

- The **Most General View** : that the hypothesis is specified by the conjunction of relations labelling its upper marks, which leads the rule to make errors of commission ; and.

[5] Here, we say that a relation is **above** a mark if it is outside the subtree dominated by that mark. Similarly, a relation is **below** a mark if it is in the subtree dominated by that mark. A relation is **between** the upper and lower marks if it is above the lower mark and below the upper mark.

- The Most Specific View : that the hypothesis is specified by the conjunction of relations labelling its lower marks, which leads the rule to make errors of omission .

For the sake of definiteness and to facilitate comparison with the last section, we will adopt the most general view. Note that this will force all negative training instances to be errors of commission rather than errors of omission. Furthermore, since root relations are always true ,we will omit them from the hypothesis. Thus the rule represented by the description space in figure 4-1 is

describe(X) & actor(X,Y) -> prefix(X,a) (iii)

rather than

describe(X) & object(X,Y) & singular(X) & indefinite(X) & past(Y)
-> prefix(X,a)

Also, for the sake of definiteness, we will assume that the rules are fired forwards. Neither of these restrictions is serious, since the algorithms for the other cases are duals of the one described below.

The partial representation of a rule provided by a rule shell is similar to the version space representation used by Mitchell et al in the Version Spaces algorithm.[6] They record two sets: S, the set of most specific rules implied by the evidence so far; and G, the set of most general rules implied by the evidence so far.

For instance, the version space corresponding to the description space in figure 4-1, is:[7]

S: {describe(X) & object(X,Y) & singular(X) & indefinite(X) & past(Y)
-> prefix(X,a)}

G: {describe(X) & actor(X,Y) -> prefix(X,a)}

---

[6] Throughout this paper, "Version Spaces" denotes the algorithm of Mitchell et al, while "version space" denotes the object.

[7] Note that we have omitted the true(X) conditions as these are always true.

The version space representation is more compact that the description space representation, but the explanation of Focussing is more messy. Version spaces do not explicitly record a piece of information vital to Version Spaces namely the correspondence between the conditions in the different rules, e.g. between object(X,Y) in S and actor(X,Y) in G.

The Brazdil/Langley/Waterman Discrimination technique of the last section corresponds to moving the upper mark down from the root to a tip of a minimal tree . We will enrich the meaning of Discrimination to cover all cases in which near/far misses cause the upper mark to descend.

The ascending of lower marks does not correspond to any technique used by Brazdil or Langley. It is done when the critic provides a positive training instance of a rule and it generalizes the hypothesis of that rule. In Winston's program, [22], it corresponds to the generalization of a concept when new examples of the concept are provided. We will call this step **Generalization** .

Focussing does not just compare the current context with a single previous context, but with all previous contexts. This is possible because all previous contexts, both selection and rejection, are summarised by the positions of the upper and lower marks in the relation trees . We need only compare the current context with the current positions of these marks. If the critic has provided us with a positive training instance then we will have a selection context, and will apply Generalization. If the critic has provided us with a commission error then we will have a rejection context , and will apply Discrimination. To some extent Generalization and Discrimination are dual processes, but this duality is not complete and the reader should beware of assuming that it is.

The description space is initialized by providing a positive training instance. For each tree in the description space, exactly one of its tip relations will be specified

by (i.e. be true in) the selection context of this instance. The lower mark is placed on this tip. The upper mark is placed on the root of the tree.

We now consider Generalization and Discrimination in more detail.

4.3.2. Generalization

The input to Generalization consists of: the selection context of a correct application of a rule: and the description space of the rule. The output consists of new lower marks for some of the trees. Each tree is considered in turn and the following steps executed.

(a) For each of the relations labelling a tip node, determine its truth value in the selection context.

(b) Exactly one of these relations will be specified by the selection context. label its node, the **current node** .

(c) Find the least upper bound of the current node and the current lower mark and make this the new lower mark.

For instance, suppose that the rule

describe(X) -> prefix(X,a)

has been correctly applied in the selection context

{dog/X, event1/Y}

and that position of the marks in the relation trees are as in figure 4-2. The tip relations which are true in the selection context are:

```
agent(dog,event1),
singular(dog),
indefinite(dog),
present(event1)
```

These specify the current nodes marked in figure 4-2, and might correspond to the child making the utterance: "A dog chases the ball". Taking the least upper bound between each current node and lower mark gives the new lower marks given

in figure 4-2. Note that the lower mark for trees 2 and 3 is unchanged, but that the lower mark of tree 1 moves to 'actor(X,Y)' and the lower mark for tree 4 moves to 'true(Y)'.

Despite these changes to the lower marks of the description space, the rule does not change form, because it is determined by the upper marks. However, Generalization does have an effect on the rule learning process, because the lifting of the lower marks can limit the choices available to Discrimination , as we will see in the next section.

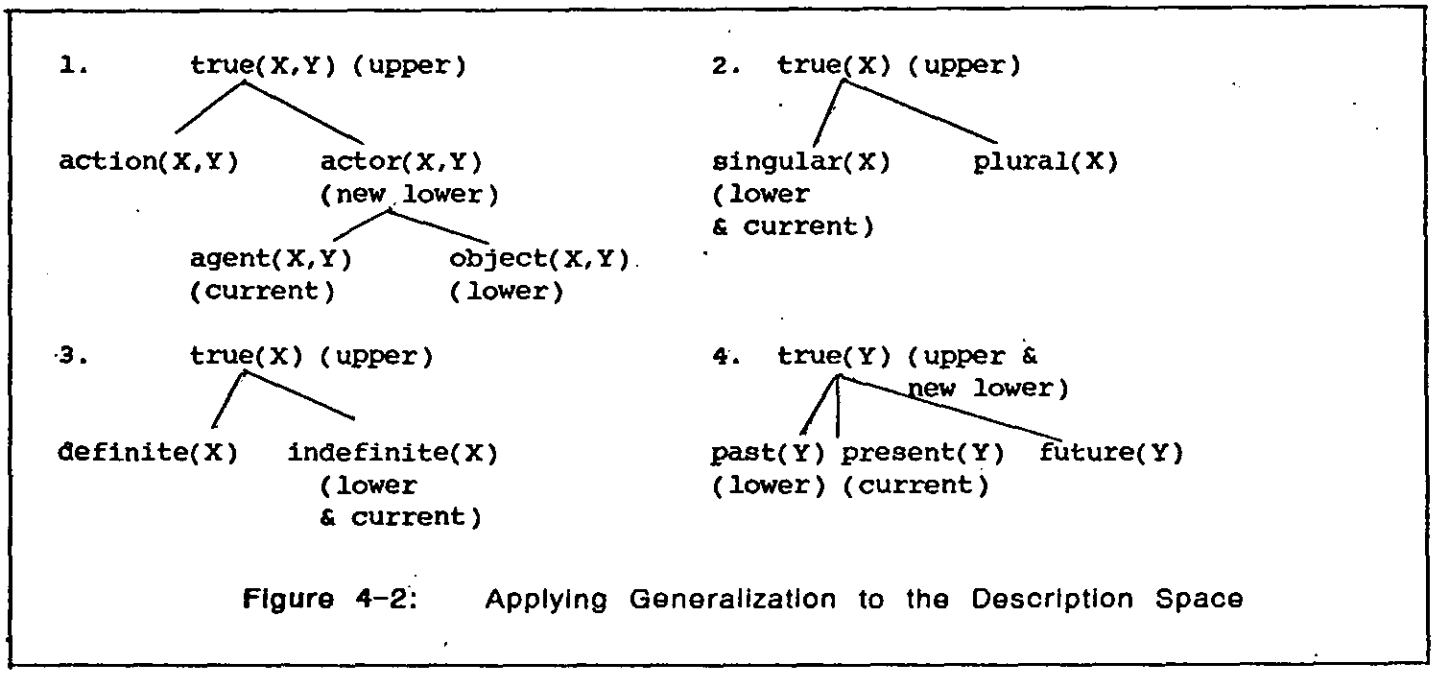


Figure 4-2: Applying Generalization to the Description Space

The version space corresponding to the new lower marks of figure 4-2 is:

S: {describe(X) & actor(X,Y) & singular(X) & indefinite(X)
-> prefix(X,a)}

G: {describe(X) -> prefix(X,a)}

4.3.3. Discrimination

The input to Discrimination consists of: the rejection context of an incorrect application of a rule; and the description space of the rule. The output consists of a new upper mark for exactly <u>one</u> of the trees.[8] Since we are dealing with a

[8] Note lack of duality.

conjunctive rule, all its conditions must be true for the rule to fire. Thus making one condition false for an instance is enough to prevent the rule firing. Each tree is considered in turn and the following steps executed.

(a) For each of the relations labelling a tip node, determine its truth value in the rejection context.

(b) Exactly one of these relations will be true in the rejection context, label its node, the current node . Note that the current node must lie below the upper mark , otherwise the rule could not have fired.

(c) If the current node lies below the lower mark then mark the tree as a **white tree** .

(d) Otherwise, the current node must lie between the upper and lower marks. Mark the tree as a **grey tree** .

At least one of the trees must be grey, otherwise the rule application would be correct. If just one tree is grey then we have a near miss . If more than one tree is grey then have a far miss . Only one of the grey trees can have its upper mark lowered. We call this grey tree the **discriminant** . Far misses can be dealt with by at least five strategies:

- **depth first** : We can pick one of the grey trees as discriminant:
- **breadth first**: Or create a new rule for each grey tree:
- **teacher option**: Or we can be told which tree to pick:
- **zero option**: Or we can do nothing.
- **avoidance option**: Or we can arrange the training order so that far misses do not arise.

Either of the first two choices may lead to the creation of rules which are over constrained and may give rise to errors of omission . Such rules should be deleted. In the case of depth first search the program should then backup and chose another discriminant. The breadth first option corresponds to Langley's solution to far misses, as described in section 4.2.1. The third choice is that

adopted by Waterman, with the relevance information indicating which grey tree to pick (see below). The fifth choice is that adopted by Winston. In fact, he made a feature of the dependence of his method on the training order. If there are enough instances to firm up all the trees, then far misses can be avoided by presenting all the positive instances first. However, presenting all the positive instances first can cause errors with some approaches to disjunctive rules (see section 6.3). Brazdil's solution cannot be adopted here without violating the relation tree representation of the rule hypothesis, but it is similar to the Version Spaces solution (see below).

Once the discriminant has been picked its upper mark is lowered, just enough to exclude the current node . This is done by setting the new upper mark to be the least upper bound of the current node and the lower mark .[9] To illustrate Discrimination suppose that the rule

describe(X) -> prefix(X,a)

has been incorrectly applied in the rejection context :

(chases/X, event2/Y)

and that the position of the marks in the relation trees are as in figure 4-2. The tip relations which are true in the rejection context are:

```
action(chases,event2),
singular(chases),
indefinite(chases),
present(event2)
```

These specify the current nodes marked in figure 4-3, and might correspond to the child making the utterance: "The dog a chases the ball". Trees 2 and 3 are white and trees 1 and 4 are grey. If tree 1 is chosen as the discriminant then action(X,Y) can be excluded by lowering the upper mark from true(X,Y) to actor(X,Y). The new rule is:

---

[9] Note lack of duality with Generalization , i.e. we do not use the greatest lower bound of the upper and current mark.

describe(X) & actor(X,Y) -> prefix(X,a) (iv)

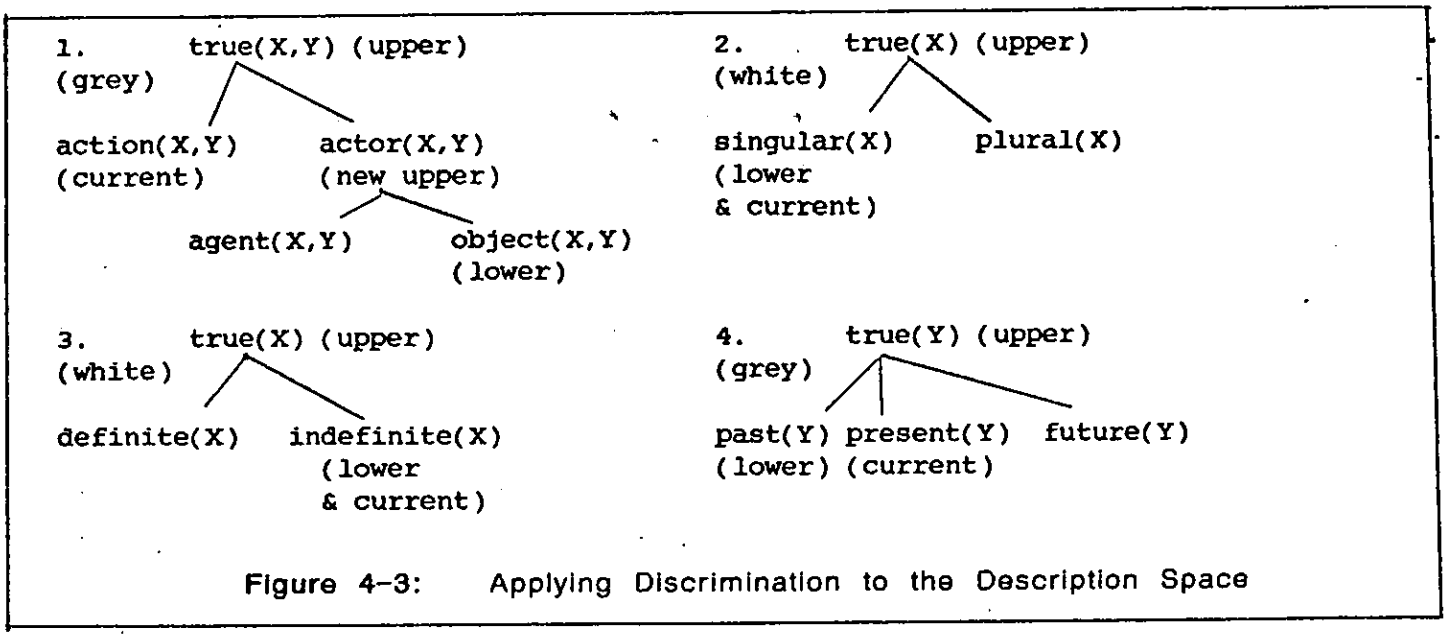


Figure 4-3: Applying Discrimination to the Description Space

4.3.4. Far Misses

If tree 4 had been picked as the discriminant then the new rule would have been:

describe(X) & past(Y) -> prefix(X,a)

Since the tense of an utterance does not affect whether the article "a" should prefix actors then this rule would eventually be guilty of an error of omission, e.g. in the context (dog/X, event4/Y), where present(event4), the rule would not fire when it should. At this stage the rule should be deleted, and if the alternative rule, (iv), has not already been formed, then it should now.

Note that if the Generalization of past(Y) and present(Y) to true(Y), had preceded the current Discrimination step, then tree 4 would not have been a grey tree and, hence, not available as the discriminant. Thus Generalization can prevent the occurrence of far misses, and the consequent creation of erroneous rules. For this reason it is best to make Generalization steps before Discrimination steps.

When Waterman's program makes a wrong decision, either a control or factual error, it is given information additional to that provided by the ideal trace

Waterman calls this information **relevance information** . This is a set of the trees that need to be considered to make a correct decision. In the situation of figure 4-3, the relevance information would indicate that tree 1 is to be considered, and that tree 4 is not.

The version space corresponding to the new upper marks of figure 4-3 is

```
S: {describe(X) & object(X,Y) & singular(X) &
      indefinite(X) & past(Y)
            -> prefix(X,a)}

G: {describe(X) & actor(X,Y) -> prefix(X,a),
      describe(X) & past(Y) -> prefix(X,a)}
```

Note that the correspondence between object(X,Y) and actor(X,Y) is not explicitly recorded and must be rederived before further Generalization/Discrimination can be applied. This is a disadvantage of the version space representation.

In this version space G is a doubleton; the two members representing the outcomes of the twofold choice of discriminant in the far miss situation. Thus the version space can simultaneously represent several alternative versions of a rule. This explains why G is a set. However, it does not explain why S is a set. Generalization never involves choices, even when the version space is representing several rules, so S will always be a singleton! Allowing S to be a set appears to be a minor flaw in the LEX program.[10]

4.3.5. Differences between Version Spaces and Focussing

Focussing and Version Spaces are similar in many ways, most importantly they both combine Generalization and Discrimination . However, there are a few differences.

- The Focussing relation trees explicitly store the correspondences between

---

[10] Mitchell, (personal communication), states that while this is the case for the rule language used in this paper, (and for the rule language used in LEX), with other rule languages, such as that used in Meta-DENDRAL, S can be a non-singleton set.

relations, and this must be rederived in Version Spaces.

- As shown in section 5.1 below, the two representations differ in the rule-creation phase. The specific boundary of the version space represents a more specific concept than that represented by the lower marks in the Focussing algorithm.

- In the Focussing algorithm, Generalization and Discrimination affect only one set of marks, the lower and upper marks respectively. Version Spaces actually requires the updating of both boundary sets for both processes. The extra operations prune the boundary sets: patterns matching negative instances are removed from the specific boundary set, and patterns <u>not</u> matching positive instances are removed from the general set. None of the examples given for LEX use these pruning operations, and they seem to be needed only when dealing with graph-like description spaces and for noisy data.

4.4. Summary of Conjunctive Modification Techniques

The following relationships hold between the techniques presented above:

- Focussing combines Generalization and Discrimination in a clean manner. They are near duals, but Discrimination is non-deterministic, whereas Generalization is not.

- Focussing is similar to Version Spaces, except that the Focussing relation trees explicitly store the correspondences between relations, and this must be rederived in Version Spaces .

- The firming up of a rule shell in Focussing and the meeting of S and G in Version Spaces, provide guarantees that the learning process has terminated. No such guarantees are provided by Generalization or Discrimination used alone.

- Focussing contains a generalization of Langley's Discrimination so that it can deal with non-minimal relation trees .

- Discrimination is a generalization of Brazdil's rule instantiation.

- Rule ordering is independent of the other modification techniques.

Discrimination on far misses introduces choice and search into the modification processes.

Two possible flaws were detected in the published research: Langley's use of rule ordering for factual faults , and Mitchell et al's use of a set of most specific rules, when there can never be more than one. Both authors have told us that while these are flaws in the programs referenced in this paper, they can be features in programs designed to tackle more difficult problems, e.g. cope with noisy data.

5. Creating New Rules

In this section we describe how new rules are created, in contrast to the modification of existing rules. Some of the programs don't address this task. None of the programs actually create new rules in the sense of deriving new conditions and conclusions. Instead, they use a degenerate form of rule modification, on a rule with no conditions.

5.1. Modifying the Empty Shell

One obvious technique for creating a rule is to treat its absence as an error of omission, and use the standard techniques to correct this error.

The idea is to modify rules that have no conditions from the description space, only a conclusion. (The lack of conditions can cause both factual and control faults .) We call such rules **empty rules** . An empty rule, together with its description space, constitute an **empty shell** . Mitchell and Shapiro adopt this approach, which we call **Modifying the Empty Shell**

In our example,

describe(X) -> prefix(X,a)

will be an empty rule. The condition describe(X) does not take part in learning, it is not in the description space.

We describe the method used by LEX, Shapiro's method is similar.

LEX creates a new rule shell when the process of Solution Extraction discovers an error of omission the first time that a new rule is used in the ideal trace . In Focussing terminology, the lower marks are initialized to the current marks, the upper marks are initialized to the roots of the trees.

If the most specific view is adopted, the rule shell implies that the rule should only be used in the current context. The most general view implies that the rule should always be used.

For example, suppose that LEX is learning when it should use the empty rule

describe(X) -> prefix(X,a).

The ideal trace uses the rule for the first time in the context

```
agent(dog,event1).
singular(dog).
indefinite(dog).
present(event1).
```

(This corresponds to e.g. "A dog chases a ball.") LEX creates the new rule shell shown in figure 5-1.

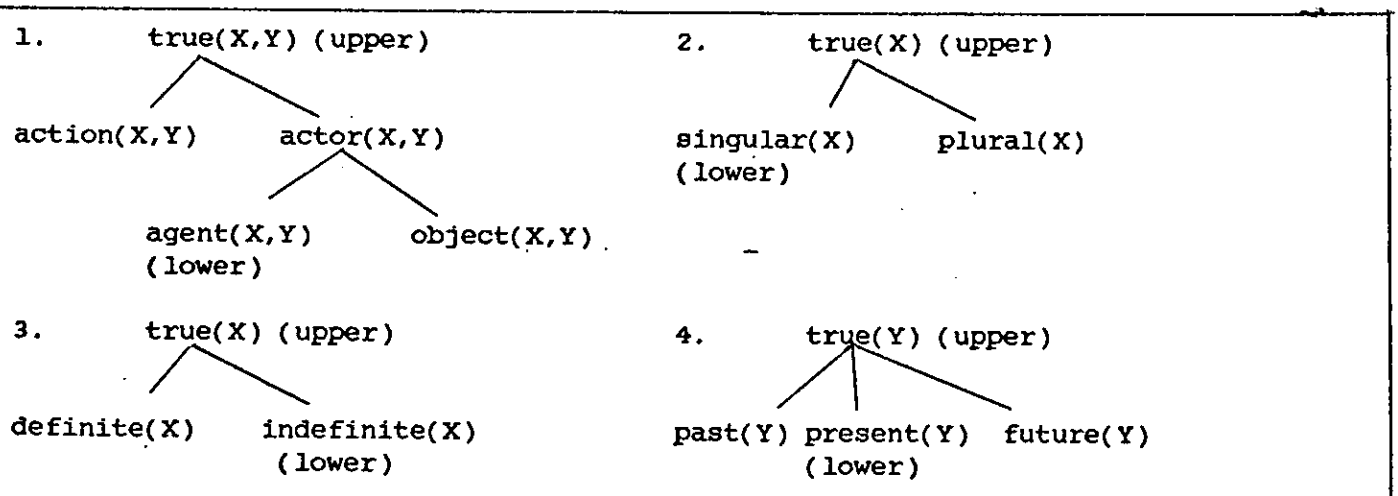


Figure 5-1: The Initialized Rule Shell

This corresponds to the version space

```
S: describe(dog) & agent(dog,event1) & singular(dog) & indefinite(dog)
                                          -> prefix(dog,a).

G: describe(X) -> prefix(X,a).
```

Note the rule S is more specific than that represented by the Focussing algorithm. The version space representation allows the representation of very specific concepts, these will usually be generalized by further examples.

Note that LEX also creates new rule-shells when the concept it is trying to learn is disjunctive, see section 6.3.

Shapiro adopts a similar approach, although he is dealing with factual faults. MIS begins with the empty program. This fails to account for a positive instance, (an error of omission), and so it is modified using his standard technique.

5.2. Guided Rule Creation

Waterman's program is given **training information** by the user[11] This information allows the program to directly construct the **training rule** , the rule that should have been used in the current situation. The program first tries to modify the existing rule set to cover the training rule, but if it is unable to do this, it adds the training rule to its set, thus obtaining a new rule. We call this approach **Guided Rule Creation** .

The addition of the training rule can lead to control faults . Waterman adopts the approach likely to minimize this possibility by adding the new rule just before the rule that fired incorrectly, correcting both an error of commission and an error of omission. The technique is rather ad-hoc, and there seems nothing to prevent further instances causing the program to loop, but Waterman does not discuss this.

---

[11] There are several different versions of Waterman's program. In some, the role of the user is played by an expert program, in another the program uses a type of database to get this information. However, for our purposes the distinction is unimportant.

Langley's Rule Creation approach is somewhat similar to Waterman's. However, AMBER obtains its training rule from a set of meta-rules rather than the user. The distinction doesn't appear to be very significant.

5.3. Summary

We have described two rule creation techniques:

- Modifying the Empty Shell
- Guided Rule Creation.

Both of these techniques are strongly dependent on the description space , which supplies the conditions of the new rule.

Modifying the Empty Shell is also strongly dependent on the ideal trace The ideal trace is needed to discover the conclusion of the new rule. Of course, it also shows that an error has occurred, and thus indicates the need for a new rule.

This technique thus combines conclusions provided by the ideal trace with conditions from the description space, apparently a rather trivial form of rule creation. However, it would perhaps be too much to expect the program to somehow discover new conditions and conclusions for itself.

Guided Rule Creation obtains the conditions and conclusions of the new rule from the user. This is really an uninteresting way of creating new rules, and Waterman's implementation may be flawed.

6. Modification Techniques for Disjunctive Rules

The techniques discussed above have all been for conjunctive rules or conjunctive concepts. In this section we consider the extension of these techniques to learn disjunctive rules and concepts, i.e. concepts that involve disjunction, as well as conjunction and negation.

In one sense the extension to disjunctive concepts is trivial. Any concept involving conjunction, negation and disjunction can be put in disjunctive normal form, i.e. rewritten into a logically equivalent form consisting of a disjunction of conjunctions of negated or unnegated atomic formulae. Each disjunct can then be learned separately using conjunctive learning techniques. From the viewpoint of rule learning this means that any disjunctive rule can be split into a number of conjunctive rules, e.g.

```
describe(X) & actor(X,Y) & [singular(X) v definite(X)]
        -> use-article(X)
```

(meaning: "Use an article when describing a singular or definite actor") is logically equivalent to:

```
describe(X) & actor(X) & singular(X) -> use-article(X)
describe(X) & actor(X) & definite(X) -> use-article(X)
```

The problem of learning disjunctive rules then becomes one of knowing when to split a rule shell into two or more rule shells, and which positive instances to associate with which shells. (Negative instances are counterexamples to all rules and apply to all shells) Since the Focussing concept learning technique subsumes the other techniques discussed above, we will compare various techniques for dealing with disjunctive concepts by considering what modifications they suggest to Focussing. The only exception to this is Quinlan's Classification, which cannot be described in terms of Focussing. We discuss this in sections 6.5 and 6.6, and compare it to Focussing in section 6.7.

Of the programs discussed above, only Langley's AMBER, Mitchell's LEX and Brazdil's ELM, attempted to deal with disjunctive concepts.

6.1. How Focussing Fails on Disjunctive Rules

What difficulties arise when Focussing is applied to a disjunctive concept?

First note that Focussing has some limited scope for disjunction. For example the rule

describe(X) & actor(X,Y) -> prefix(X,a)

effectively represents a rule containing an (exclusive) disjunction:

describe(X) & (agent(X,Y) v object(X,Y)) -> prefix(X,a).

However, concepts containing disjunctions between trees cannot be represented. An example of such a rule is:

```
describe(X) & actor(X,Y) & (singular(X) v definite(X))
        -> use-article(X).
```

We now consider how Focussing might be adapted to learn this rule. We use the same description space as before. Suppose that we have the instances given in figure 6-1.

```
(a)  agent(dog,event1),              (b) object(ball,event2),
     singular(dog),                      singular(ball),
     indefinite(dog),                    indefinite(ball),
     present(event1).                    past(event2).

(c)  agent(dogs,event3),             (d) object(balls,event4),
     plural(dogs),                       plural(balls),
     indefinite(dogs),                   definite(balls),
     present(event3).                    past(event4).

(e)  object(ball,event5),            (f) action(chases,event6),
     singular(ball),                     singular(chases),
     definite(ball),                     definite(chases),
     present(event5).                    present(event6).
```

Figure 6-1: Training Instances for a Disjunctive Concept

Instances (a), (b), (d) and (e) are positive training instances , (c) and (f) are negative ones. They might correspond to the child making the utterances:

(a) "A dog chases ...."

(b) ".... chased a ball"

(c) "a dogs chases ...."

(d) ".... chased the balls"

(e) ".... chases the ball."

(f) ".... the chases ...."

Suppose the training order is instance (a), followed by instance (b). The description space will now contain the marks shown in figure 6-2. Tree 4 is firmed up.

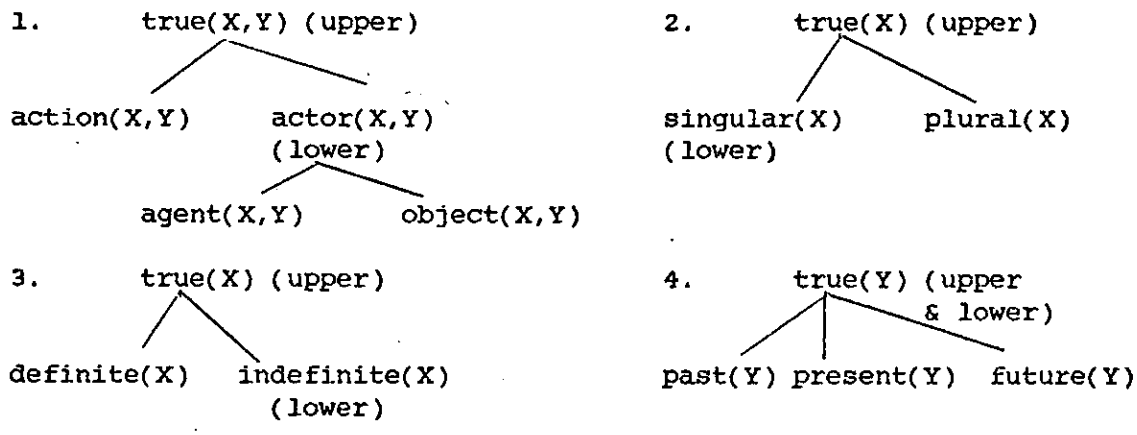


Figure 6-2: The Description Space After Two Positive Instances

Now instance (c) is presented. This is a negative instance, so Discrimination occurs. Only Tree 2 is grey, so it becomes the discriminant and is firmed up, while trees 1 and 3 remain unchanged. The situation is shown in figure 6-3.

Suppose that we now present the instance (d), a positive instance. Focussing fails because the current instance on tree 2 is above the upper mark, but the instance is positive. If the instances were presented in another order, different behaviour would be produced, but a similar problem would always occur. For example, if the first three instances were (a), (b) and (d), trees 2, 3 and 4 all become firmed up. Then instance (c) causes the algorithm to fail as it is below

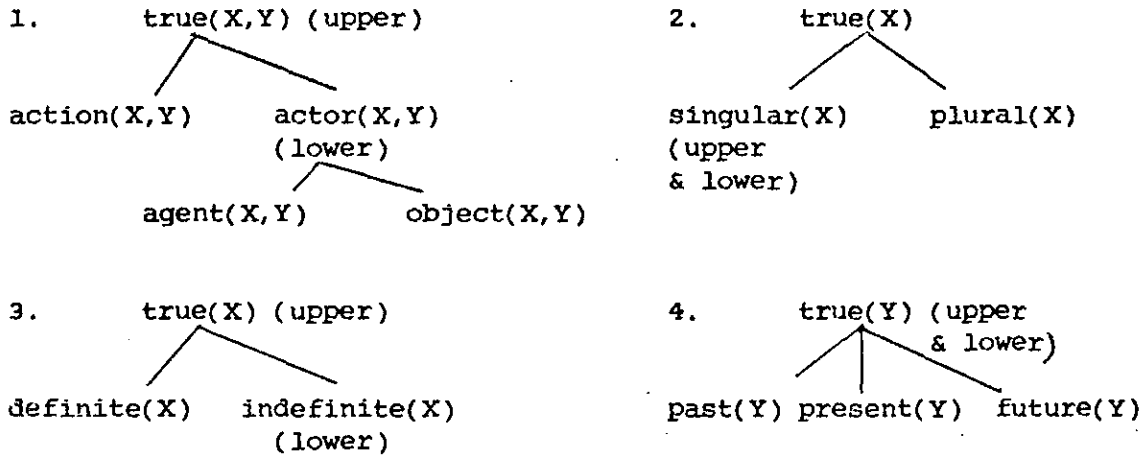


Figure 6-3: The Description Space After a Negative Instance

the lower mark on all four trees.

6.2. Other Causes of Inconsistencies in Focussing

Unfortunately, it is not just disjunctive concepts that can cause such inconsistencies. Each of the other possible causes suggests different ways of repairing the situation, contributing to a combinatorial explosion in the learning process.

One obvious cause is noisy data. The positive instance that appears above the upper mark, or the negative instance that appears below the lower mark may just be wrongly classified. The solution in this case is just to ignore the evidence, and continue. Possibly, the noisy data occurred earlier in the learning process, in which case the solution is to back up and ignore the earlier evidence.

A cause we have already met (see section 4.2.1) is a wrong choice of discriminant when discriminating against a far miss . The solution in this case is to back up to the choice point and choose another grey tree as discriminant.

An inconsistency can also be caused by an inadequate description space . For instance, suppose the correct form of the rule is:

```
describe(X) action(X,Y) & [present(Y) v future(Y)]
        -> use-present(X)
```

Since the tense tree is ternary (see figure 6-4) then positive instances for 'present' and 'future' will cause Generalization to move the lower bound to the root node.

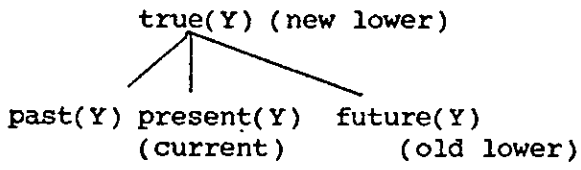


Figure 6-4: The Tense Tree Before Manipulation

A negative instance for 'past' will now be below the lower bound and hence cause an inconsistency. The solution in this case is to manipulate the tense tree into the form given in figure 6-5.

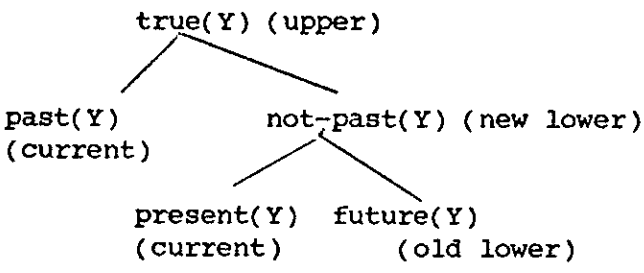


Figure 6-5: The Tense Tree After Manipulation

We have developed a technique to do this which we call **Tree Hacking** . It can be summarised as follows:

(a) Mark each tip that has been specified by a positive training instance with a +. If the tree has ever been used as a discriminant for a negative training instance, then mark the tip specified with a −. (NB these marks will be inherently contradictory.) Mark any unmarked nodes either + or −, nondeterministically.

(b) Remove from the tree all arcs and all nodes except the root and tip nodes.

(c) If there is only one + node, then join this to the root node. Otherwise, create a new node named P, say, and join it to the root and all + nodes to it.

(d) If there is only one - node, then join this to the root node. Otherwise, create a new node named N, say, and join it to the root and all - nodes to it.

This procedure is correct, but might be improved by making it preserve any of the existing structure of the tree which does not need to be altered.

The description space may also be inadequate because a relation is missing. For instance, suppose the tense tree is missing, but a positive and negative instance differ only in the specification of present(Y) in the positive instance and past(Y) in the negative instance. The two instances will specify identical marks in the relation trees, and thus are bound to cause an inconsistency. The solution in this case is to create a new relation tree, but we know of no technique for doing this.

Note that each of these possible causes of what might be identical looking inconsistencies, suggests a different solution. Thus an inconsistency causes a choice point in the learning process, leading to search and a possible combinatorial explosion. In the rest of this section we will assume that inconsistencies are caused only by disjunctive rules.

Note that the detection of inconsistencies is only possible when the learning technique combines Generalization and Discrimination, and not when one of these techniques is used on its own. Since Langley used Discrimination without Generalization, his program had to adopt a different method of creating disjunctive rules.

6.3. Shell Creation: The Disjunctive Technique of Mitchell et al

To cope with inconsistencies caused by disjunctive rules it is necessary to introduce a new rule shell and to divide the positive instances between the old and the new shells. Negative instances should apply to both shells. Both rule shells have the same basis, e.g. prefix(X,a), but will firm up to different hypotheses, e.g. present(X) and future(X). Mitchell et al, [15], have implemented a technique

for doing this for version spaces, which can be easily adapted to Focussing. We call it **Shell Creation** . However, it seems to suffer from some serious flaws. We describe Shell Creation in this section and discuss its flaws.

LEX detects an inconsistency by the occurrence of a positive instance that is excluded by the version space . In Focussing, this corresponds to a positive instance above one of the upper mark, as illustrated above. The other possible inconsistency illustrated above, a negative instance that is included by the version space, cannot be dealt with by introducing a new rule shell. This is because negative instances apply to both rule shells, and the offending negative instance will continue to cause an inconsistency in the old shell.

Mitchell et al assume that such inconsistencies are caused by a disjunctive concept, rather than: poor choices during far miss Discrimination, noisy data or an inadequate description space.

Shell Creation creates a new rule shell, represented by a new version space, that accounts for the positive instance. The S set of the new version space is the positive training instance. The description given doesn't tell us what the G set should be. If we adopt the procedure used by Mitchell et al for creating new rules, (see section 5), G is the most general set. Note that the G set of the old version space can't be used, as the current positive training instance lies above it.

In Focussing this means that the offending positive training instance is used to set the lower marks of the new rule shell, and the upper marks are set to be the roots of the trees.

How are the subsequent positive instances divided between the old and new rule shells?

Mitchell et al write:

> "This new heuristic [rule] will be updated by all subsequent negative instances associated with operator O, and by any subsequent positive instances associated with operator O and to which at least some member of its version space applies."

By "operator O", they mean the shared conclusion of the two rule shells.

Note that the new shell gets preferential treatment when it comes to allocating the new positive instances between the shells, whereas the old shell gets preferential treatment when it comes to allocating the old positive instances. Negative instances, of course, apply to both shells.

Even if LEX does eventually learn the disjunctive rule, which is by no means certain, using Shell Creation can cause errors during training. This is because the upper marks of the new rule shell might allow previous negative instances, so the rule might be used in a situation that is already known to be a negative training instance! However, this is not the major flaw.

There are several related major flaws with Shell Creation .

1. The old rule shell may contain a mixture of positive instances, relating to both rules. When the new rule shell is formed it will be missing some of these positive instances, and Version Spaces will under-generalize. Hence, some of the trees may never be firmed up.

2. The inclusion in the old rule shell of positive instances properly belonging to the new shell will cause Version Spaces to over-generalize the old shell. This will cause inconsistencies of the kind not mentioned by Mitchell et al; negative instances will be accepted by the version space .

3. The new rule shell may be credited with positive instances which should have been credited to the old rule shell. This will cause the old shell to be undergeneralized and the new shell to be over-generalized.

We will demonstrate the problems by adapting Shell Creation to Focussing .

Shell Creation is very sensitive to the training order. If, in our running example, the instances are presented in the order (a), (b), (c), (d), (e), (f),

then it works correctly. After (a), (b) and (c), the situation is as shown in figure 6-3. Then, just as explained in section 6.1, an inconsistency occurs when positive instance (d) is presented - the current mark is already excluded by the upper mark in tree 2.

Shell Creation now suggests creating a new rule shell, using (d) to determine the lower marks, and putting the upper marks on the roots of the relation trees. This is shown in figure 6-6.

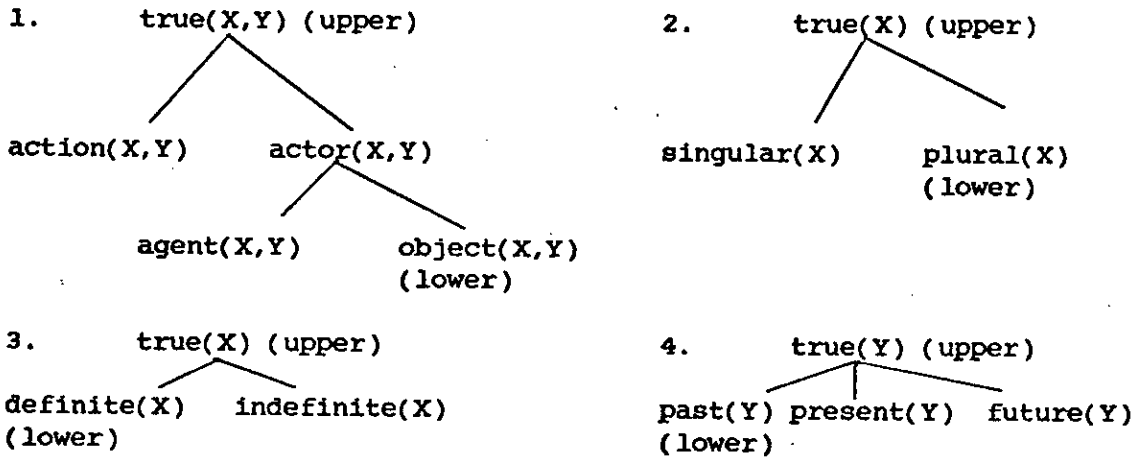


Figure 6-6: The New Description Space Following Inconsistency Detection

The instances (e) and (f) are now presented to this rule shell. These firm up trees 1, 2 and 4. If the negative instance (c) were now presented to the new shell, then tree 3 would also be firmed up and the new rule shell describes the rule:

describe(X) & actor(X,Y) & definite(X) -> use-action(X)

However, Shell Creation does not provide for the re-presentation of negative instances. (f) should also be presented to the old shell, as it is a negative instance, and such instances are negative for all shells. Also, (e) should be re-presented to the old shell, as it is a positive instance relevant to both shells. However, Shell Creation does not provide for either.

The new shell is not created at all if the instances are presented in the order (a), (b), (d), (c), (e), (f), with the positive instance (d) before the negative one (c). After the first two instances the situation is that shown in figure 6-2 above. The next positive instance, (d), causes Generalization to produce the situation shown in figure 6-7.

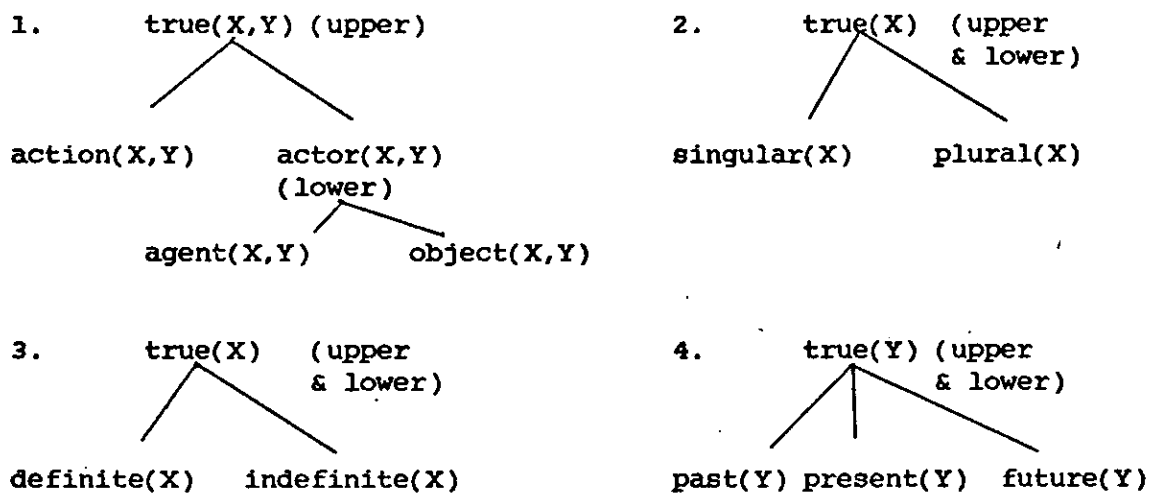


Figure 6-7: The New Description Space After Three Positive Instances

Trees 2, 3 and 4 are all firmed up at the true root. The next negative instance (c) would cause the lower and upper marks to cross. In Version Spaces, this corresponds to members of the S set becoming more general than members of the G set. As explained above, this kind of inconsistency cannot be dealt with by creating a new rule shell.

Note that there was no opportunity to take a copy of the description space to produce a second rule. The two kinds of positive instances, those due to singular(X) and those due to definite(X), are both mixed in the one rule shell. If we consider that the first copy of the space should learn the singular(X) part, the positive instance (d) due to the presence of definite(X) can be considered "false". The lower marks are raised incorrectly in order to include this false instance.

Negative instances are not such a problem, because they are negative instances

for all rule shells. However, in Shell Creation the negative instances which are given before the creation of the new rule shell do not get an opportunity to influence it. To overcome this problem, all old negative instances should be kept, and all new copies of the rule shell should be updated with them. This is not enough, however. All the positive instances have to be kept as well!

When the a new rule shell is formed the false positive instances should be removed from the old shell. There is no simple way to do this. A flawless solution requires that all instances both positive and negative, are stored. We outline such an extension to Focussing below, which we will call **Refocussing** .

The first time an inconsistency is detected:

- a copy of the rule shell must be made, and the Focussing process must be restarted using the negative instances on both shells and the positive instances on only one shell.

- On each iteration of generalization, the positive instance should be allocated to a shell so as to avoid an inconsistency, if possible.

- Choice points must be saved for subsequent back-up, and back-up to re-allocate the positive instances should be the first option if subsequent inconsistencies are detected.

- Failing this, a further subdivision of the rule shell can be made, and the Focussing process restarted again.

This process seems very inefficient, but something like it appears to be a requirement for all systems that learn disjunctive concepts.[12] See sections 6.5 and 6.7 for further discussion.

In conclusion, Mitchell et al have described how Version Spaces can be modified

---

[12] Refocussing is somewhat similar to another technique of Mitchell, described in [11], and Iba, [6], placed in a focussing context. Like Refocussing, the methods are computationally expensive, and require that all data must be kept.

to learn disjunctive concepts using Shell Creation. We have shown how Focussing can be similarly adapted. However, these "solutions" are far from perfect, and rely on very favourable training orders. It appears that any adaptation of Focussing to learn disjunctive concepts correctly, e.g. Refocussing above, must include storage of all the training instances.

6.4. Shell Forking: The Disjunctive Technique of Langley and Brazdil

If the rule being learned is disjunctive then certain training orders of the instances can cause a far miss to occur. Langley's and Brazdil's method of dealing with far misses will then create a disjunctive rule. We call this technique Shell Forking . To force such a far miss a positive instance must be given which is true for both disjuncts, followed by a negative instance. Langley's version of Shell Forking will then make two copies of the rule shell, and Brazdil's version will put a disjunct in the hypothesis. However, the technique is very training order dependent. If a positive instance is given which is true in only one disjunct, then the far miss will never occur. In addition, all the caveats given in the last section about the proper division of the positive instances between rule shells, also apply here.

We can illustrate Shell Forking using our running example. A favourable training order is (e) followed by (c). Figure 6-8 shows the situation after positive instance (e). Presenting the negative instance (c) produces a far miss . Discrimination can be applied in three ways, as shown be the new upper marks in the figure. Trees 1, 2 and 3 are grey. In Langley's formulation of Discrimination this suggests division of the existing rule shell into three shells, corresponding to the rules:

(i) describe(X) & object(X,Y) -> use-article(X)

(ii) describe(X) & singular(X) -> use-article(X)

(iii) describe(X) & definite(X) -> use-article(X)

Two of these, (ii) and (iii), are the representation of the disjunctive rule that

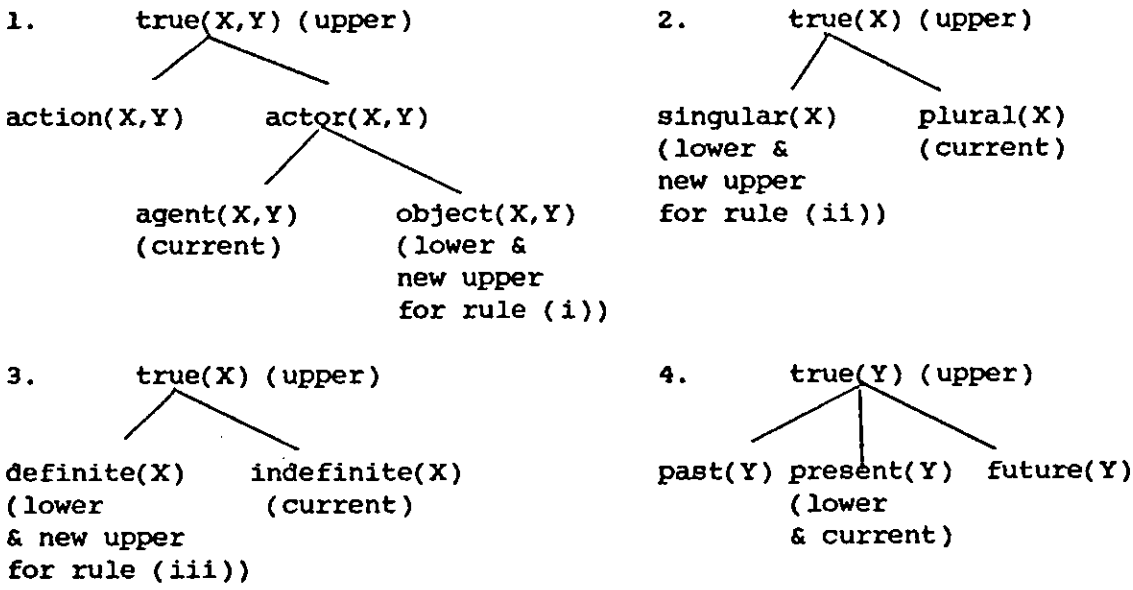


Figure 6-8: Far Misses Indicate Some Possible Disjunctive Rules

we want. The remaining rule, (I), is wrong. In the shell corresponding to this rule, tree 1 has firmed up at object(X,Y), instead of at actor(X,Y). This rule is an accident, which may or may not be downgraded by receiving a low numeric score.

In Langley's program, these three rules will be thrown into the pool where their numerical strength will be determined by their subsequent success in prediction. This is also dependent on training order. If a large sequence of plural, definite, object, instances is given then the singular rule will be discriminated against. If a large sequence of singular, definite, agent, instances is given, then the object rule will be discriminated against. We want the latter to happen and the former not to happen, but there is no way of guaranteeing this.

Langley's program doesn't have to keep all the data. However, it does keep all the rules! The possible haphazard nature of the learning process might be avoided if all the data was preserved, with some modification of the rule-ordering technique that takes into account the proportion of data explained.

Although Brazdil's technique creates disjunctive rules to deal with far misses (as

explained in section 4.2.1), this is only a temporary expedient. The technique is based on the assumption that the rule is really conjunctive, and, at the first opportunity, all but one of the disjuncts is pruned. Thus Brazdil's program does not really learn disjunctive concepts. It is possible that the program could be adapted to learn disjunctive concepts, but it might then suffer on conjunctive concepts.

6.5. Classification

None of the programs discussed above can deal flawlessly with disjunctive concepts. However, there are concept learning programs which can do so, for instance, ID3, [17] and Iba's program, [6]. In this section we describe Quinlan's ID3, and in the next section contrast it with Focussing.

ID3 can be fitted into the Focussing framework when it is working on conjunctive concepts. However, to discuss disjunction, we must depart from this framework, and describe the technique used by ID3, which we will call **Classification** .

Classification differs from Focussing in many important respects. Firstly, it tests each relation in turn on <u>all</u> the training instances, whereas Focussing tests each instance in turn, on all the relations. Classification keeps all its data, whereas Focussing does not. Keeping all the data seems to be a necessary feature of a flawless learning technique for disjunctive concepts.

Classification represents its concept via a **decision tree** . Each node of this tree is labelled by an **attribute** , and the branches below this are labelled by the different possible values of this attribute (see figure 6-9). An attribute in Classification corresponds to a Focussing relation tree of depth one. Each value of the attribute corresponds to a branch of the relation tree. For instance, the attribute 'tense' might take values: past, present and future, which corresponds to the tense relation tree (see figure 6-4).

The decision tree is a representation of the partially learned concept. The tree

is initially empty, and Classification grows the tree. This involves the following steps:

(a) If all the instances in the set are positive, then a node is created marked as being within the concept. If they are all negative, then a node is created and marked as being outside the concept. If there are no instances in the set then a node is created and marked arbitrarily as within or outside the concept. In any of these cases the process then halts.

(b) Otherwise, an attribute is chosen in a heuristic manner. (Quinlan uses an information theoretic method.) A new decision node is created. The node is marked with the name of the attribute, and has a daughter node for each of the possible values that the attribute can take.

(c) The training instances are partitioned into subsets according to the values of the attribute. For attribute A, we call this step **splitting** on A.

(d) The process is applied recursively to each of the subsets.

6.6. How Classification Handles Disjunction

We now show how Quinlan's program is able to learn disjunctive rules. Consider the problem defined by instances given in figure 6-1 above using the attributes adapted from the relation trees in figure 4-1.

Each of the relation trees in figure 4-1 will give rise to an attribute . Tree 2 gives an attribute number(X), with values: singular(X) or plural(X). Tree 3 gives the attribute definiteness(X) with values: definite(X) and indefinite(X). Tree 4 gives the attribute tense(Y) with values: past(Y), present(Y) and future(Y). Tree 1 must first be flattened to a tree of depth 1, and then gives an attribute case(X,Y), with values: action(X,Y), agent(X,Y) and object(X,Y). Note that this process destroys the structure of tree 1.

If Classification first splits on the number attribute, the singular subset contains only positive instances, so this branch is complete. The plural subset contains both

types of instances, and more Classification needs to be done. Splitting on the definite/indefinite attribute then completes the process, forming the decision tree in figure 6-9.

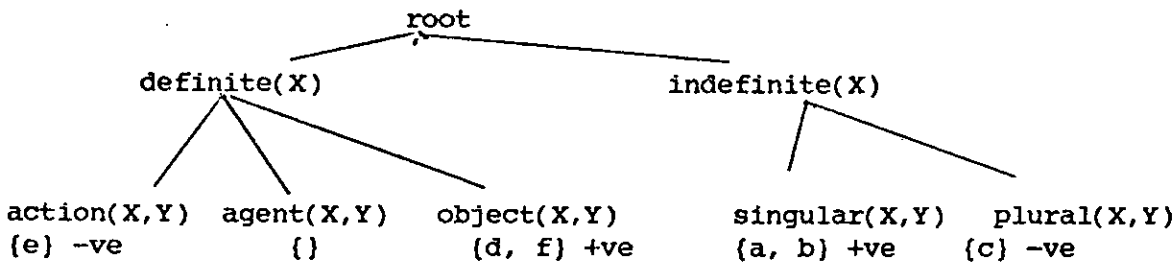


Figure 6-9: A Decision Tree

The tip nodes are labelled with the set of instances that they represent, and whether the instances are positive or negative. This tree represents the rule

```
describe(X) &
((definite(X) & object(X)) v (indefinite(X) & singular(X)))
        -> use-article(X).                                          (v)
```

The rule (v) is not quite the one we intended to learn, although it will always give the correct results. There is no splitting order which will give us the form of rule we intended. The reason is:

1. That some of the logical properties represented in the case relation tree were lost when it was flattened to depth one.

2. That Classification is sometimes forced to include irrelevant attributes on one branch in order that they can appear on another branch, e.g. the number attribute above.

However, a poor splitting heuristic can make the problem much worse. For example, if the tense attribute had been chosen first, then the hypothesis of rule (v) would be duplicated as two disjuncts: one headed by present(X) and one by past(X), which is completely redundant. Quinlan's information theoretic splitting heuristic is quite good at avoiding such redundancy where possible, but it is not perfect.

6.7. A Comparison of Focussing and Classification

In this section we compare the performance of Focussing and Classification on conjunctive and disjunctive concepts.

Because it uses both Generalization and Discrimination, Focussing is able to detect that a rule has been completely learned, because all the trees are firmed up. Classification cannot guarantee its output in this way. Further training instances might always cause Classification to refine the rule further, splitting on a tip node which previously only contained instances of one type, but now contains mixed instances.

Another drawback of Classification is that the decision trees it produces are often non-optimal. Classification will usually produce a disjunctive rule, even when Focussing would produce a conjunctive rule on the same data. The Classification decision tree will often contain attributes that are irrelevant to the concept being learned. Quinlan's use of an information theoretic technique to choose which attribute to split on, tends to keep down the number of irrelevant attributes in the decision tree, but it is not perfect and does not exclude them all. As we have seen, even an optimal splitting order does not produce an optimal rule hypothesis.

Classification needs to have access to all the training instances before it can learn the concept. Focussing incorporates all the information, contained in a training instance, into the current rule shell, and then discards it. Not only does this save storage space, but it means that it can use the partially learned rule before all the instances have been provided. However, the discussion above suggests that in order to learn *disjunctive* concepts, Focussing, or any other concept learning technique, would have to retain all the instances, and might have to rebuild structure.

Classification, as we have described it above, does not need to rebuild structure.

Quinlan describes in [17] how Classification can be adapted to learn where it only has access to part of the data at any time. In this case, the decision tree must be rebuilt when necessary. Iba, [6], describes a program that can learn disjunctive concepts. Iba's program also spends much of the time rebuilding structures, and it keeps all its training instances for this purpose.

Against these disadvantages, Classification is able to learn disjunctive rules and to avoid having to confront far misses. It is best for Focussing to generalize from all the positive instances before discriminating from any negative instances. Classification is not affected by the training order of instances, although the simplicity of its final decision tree is dependent on the order in which it splits on attributes. Further, while ordering positive before negative instances may enable Focussing to avoid far misses, it makes it more susceptible to errors when using Shell Creation to learn disjunctive concepts.

The comparison of Classification with Focussing shows that while Classification can learn both conjunctive and disjunctive concepts, the standard Focussing algorithm can only learn conjunctive concepts. On conjunctive concepts, however, Focussing has several advantages. seen above, the rules produced by Focussing can be much simpler than the decision trees formed by Classification. Also, Focussing can delete data once it has been used, Classification cannot do this.

More details of the comparison between Classification and Focussing can be found in [16].

7. Conclusion

In this paper we have compared a collection of AI rule learning programs. Despite apparent differences of notation and terminology these programs are tackling similar problems in similar ways. Each of the programs consist of two main parts: a critic for identifying faults; and a modifier for correcting faults.

This analytic comparison set out to: abstract some rule learning techniques from the programs they were developed in; to identify their range; to locate and repair their flaws; to establish the relationship between them; and to extend their range. In this final section we summarise what we have achieved.

7.1. Techniques Abstracted

In this paper we have abstracted the following rule learning techniques. Some of them have, of course, been previously abstracted by other authors, but some abstractions are original to this paper.

- **Criticism Techniques**: Ideal Trace, Solution Extraction and Contradiction Backtracing. Contradiction Backtracing is only suitable for finding factual faults. Ideal Trace is suitable for both control and factual faults. Solution Extraction is a technique for automatically obtaining ideal traces.

- **Conjunctive Modification Techniques**: Rule Ordering, Instantiation, Discrimination, Generalization, Version Spaces and Focussing. Focussing and Version Spaces are very similar, and are suitable for both factual and control faults. They combine Discrimination and Generalization. Discrimination, in turn, subsumes Instantiation. Rule Ordering is only suitable for control faults.

- **Disjunctive Modification Techniques**: Shell Creation, Shell Forking, Refocussing and Classification. Classification is an unflawed disjunctive learning technique. Shell Creation is a flawed attempt to modify Version Spaces to learn disjunctive rules. Shell Forking is a flawed attempt to modify Discrimination. We propose Refocussing as an unflawed modification of Focussing for disjunctive rules.

- **Rule Creation Techniques**: Modifying the Empty Rule, Guided Rule Creation. Modifying the Empty Rule is an unflawed technique. Guided Rule Creation is a potentially unflawed technique, but Waterman's implementation may be flawed as he doesn't seem to address the problem of rule ordering correctly.

- **Description Space Modification**: Tree Hacking is a proposed technique for restructuring the Focussing description space in the face of apparently contradictory instances.

7.2. Flaws Indentified

We have identified a number of flaws in these techniques or in the way they have been used in the programs we studied.

- Langley used Rule Ordering for suppressing factual faults. He claims that this technique can help cope with noisy data.

- Mitchell et al kept a set of lower bounds in each version space , whereas only one lower bound will ever be needed in the situation that LEX was dealing with. Mitchell claims that a set of lower bounds *is* required where the relations in a version space form a lattice rather than a tree.

- Both Mitchell et al's Shell Creation and Langley's Shell Forking are dependent on the training order, and can fail if this is unfavourable.

7.3. Discussion

Solution Extraction and Contradiction Backtracing, are interesting new criticism techniques, which are not in widespread use yet.

Surprisingly, most of the techniques for correcting faults are equally applicable to factual and control faults. This is a consequence of the common technique of including control information in the rules, as extra conditions or instantiations, as if it were factual information.

If the rules to be learnt are known to be conjunctive, then Focussing and Version Spaces are the most powerful modification techniques. Not only do they subsume most of the other conjunctive learning techniques, but they produce a simpler solution more efficiently than Classification. They do not require instances to be stored. Focussing emerges as one of the more powerful techniques of the paper, and clearly deserves more attention than it has attracted in the past.[13].

---

[13] This lack of attention is partly the fault of the Young et al, who have only reported it in a cryptic one page paper

If the rules may be disjunctive then Classification . Refocussing or some similar technique must be used. It will be necessary to store all instances, and either to put off learning until all the instances are known or to be prepared to restart the learning process when apparently contradictory instances are input.

All the techniques described in this paper are dependant for their success on the user-supplied, description space . For instance, if the Focussing description space should contain surplus relation trees then the learning process will require extra instances and more time, and may get distracted by irrelevant far misses. If vital relation trees should be missing or be the wrong shape, then the description space will become contradictory and the learning process will fail. Automatic provision or modification of the description space is the most urgent open problem facing automatic learning. Our new Tree Hacking technique is a small contribution to the solution of this problem. Lenat's and Mitchell's recent work also offers an interesting approach to the problem. [8, 9, 12, 13].

References

[1] Brazdil, P.
Experimental Learning Model.
In *AISB/GI*, pages 46–50. Society for the Study of Artificial Intelligence and Simulation of Behaviour, 1978.

[2] Brazdil, P.
A model for error detection and correction.
PhD thesis, University of Edinburgh, 1981.

[3] Clocksin, W.F. and Mellish, C.S.
Programming in Prolog.
Springer Verlag, 1981.

[4] Dietterich, T.G., London, R., Clarkson, K. and Dromey, G.
Learning and Inductive Inference.
In Cohen, P.R. and Feigenbaum, E.A. (editors), *The Handbook of Artificial Intelligence, Volume 3*, chapter XIV. Pitman Books Ltd, 1982.

[5] Hunt, E.B., Marin, J., and Stone, P.J.
Experiments in Induction.
Academic Press, 1966.

[6] Iba, G.A.
Learning disjunctive concepts from examples.
Master's thesis, M.I.T., 1979.
Also available as AI memo 548.

[7] Langley, P.
Language acquistion through error recovery.
CIP Working Paper 432, Carnegie-Mellon University, June, 1981.

[8] Lenat, D.B.
Theory Formation by Heuristic Search. The Nature of Heuristics II: Background and Examples.
Artificial Intelligence 21(1-2):31-60, March, 1983.

[9] Lenat, D.B.
EURISKO: A Program That Learns New Heuristics and Domain Concepts. The Nature of Heuristics III: Program Design and Results.
Artificial Intelligence 21(1-2):61-98, March, 1983.

[10] Minsky, M.
Steps towards Artificial Intelligence.
In Feigenbaum, E.A. and Feldman, J. (editors), *Computers and Thought*, pages 406-450. McGraw-Hill, 1963.

[11] Mitchell, T.M.
Version Spaces: An approach to concept learning.
PhD thesis, Stanford University, 1978.

[12] Mitchell, T.M.
Toward Combining Empirical and Analytical Methods For Inferring Heuristics.
Technical Report LCSR-TR-27, Laboratory for Computer Science Research, Rutgers University, 1982.

[13] Mitchell, T.M.
Learning and Problem Solving.
In Bundy, A. (editor), *Proceedings of the Eighth IJCAI*, pages 1139-1151. International Joint Conference on Artificial Intelligence, 1983.

[14] Mitchell, T.M., Utgoff, P.E., Nudel, B. and Banerji, R.
Learning problem-solving heuristics through practice.
In *Proceedings of IJCAI-81*, pages 127-134. International Joint Conference on Artificial Intelligence, 1981.

[15] Mitchell, T.M., Utgoff, P.E. and Banerji, R.
Learning by Experimentation: Acquiring and modifying problem-solving heuristics.
In Michalski, R.S, Carbonell, J.F. and Mitchell, T.M. (editors), *Machine Learning*, pages 163-190. Tioga Press, 1983.

[16] Plummer, D.
Two techniques for Inductive Learning: A Comparison.
Research Paper 186, Dept. of Artificial Intelligence, Edinburgh, March, 1983.

[17] Quinlan, J.R.
Discovering Rules by Induction from Large Collections of Examples.
In Michie D (editor), *Expert Systems in the Micro-Electronic Age*, pages 168-201. Edinburgh University Press, 1979.

[18] Shapiro, E.
An algorithm that infers theories from facts.
In *Proceedings of IJCAI-81*, pages 446-451. International Joint Conference on Artificial Intelligence, 1981.

[19] Silver, B.
Learning Equation Solving Methods from Worked Examples.
In Michalski, R.S (editor), *Proceedings of the International Machine Learning Workshop*, pages 99-104. University of Illinois, June, 1983.
Also available from Edinburgh as Research Paper 188.

[20] Smith, R.G., Mitchell, T.M., Chestek, R.A., and Buchanan, B.G.
A model for Learning Systems.
In Reddy, R. (editor), *Proceedings of IJCAI-77*, pages 338-343. International Joint Conference on Artificial Intelligence, 1977.

[21] Waterman, D.A.
Generalization Learning Techniques for Automating the Learning of Heuristics.
Artificial Intelligence 1(1-2):121-170, 1970.

[22] Winston, P.
Learning structural descriptions from examples.
In Winston, P.H. (editor), *The psychology of computer vision*. McGraw Hill, 1975.

[23] Young, R.M., Plotkin, G.D. and Linz, R.F.
Analysis of an extended concept-learning task.
In Reddy, R. (editor), *Proceedings of IJCAI-77*, pages 285. International Joint Conference on Artificial Intelligence, 1977.

Index of Definitions